A Logic Programming Approach to Channel Routing

Neng-Fa Zhou
zhou@mse.kyutech.ac.jp
Faculty of Computer Science and Systems Engineering
Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, 820 Fukuoka

Abstract

Channel routing is one of the key phases in VLSI design. The problem is to find routing paths among a group of terminals that satisfy a given connection requirement without overlapping each other. It is very simple to describe this problem in a logic programming language like Prolog. However, as the problem is NP-hard, it is not feasible to use simple backtracking algorithms in case the problem is big. In this paper, we present a program for the problem in Beta-Prolog, an extended Prolog that supports state tables. The problem is treated as a constraint satisfaction problem. The domains of variables and constraints are represented as state tables. By using operations over the state tables, the manipulations required on the domains and constraints are described straightforwardly. Several solutions that are comparable in quality to those obtained by special routers are described for the Deutsch's difficult problem obtained by the program on a single workstation as well as 16 workstations.

1 Introduction

Given a channel that consists of two parallel horizontal rows with terminals on them and a set of nets each specifying terminals that must be interconnected through a routing path, the channel routing problem is to find routing paths for the nets in the channel such that no paths overlap each other [2]. There are a lot of different definitions of the problem that impose different restrictions on the channel and routing paths. The problem has been studied extensively in the VLSI design community since Hashimoto and Stevens [7] proposed it in 1971. Lapaugh has proved that the problem is NP-hard [9]. Many algorithms have been proposed for the problem [3, 5, 11, 14, 16]. Most of the traditional algorithms are based on graph algorithms. Recently, several modern heuristic search algorithms, for example, neural networks [14] and genetic algorithms [11], have been proposed.

The channel routing problem is a finite-domain constraint satisfaction problem (CSP) [12]. Consider, for example, the *dogleg-free multi-layer* channel routing problem where the routing path for every net consists of only one line segment parallel to the two rows and several line segments perpendicular to the two rows, and the routing area in the channel is divided into several layers for horizontal segments and several layers for vertical segments. For each net, we need to determine the layer and the track on which the horizontal segment lies. For this problem, each net to be routed can be treated as a variable whose domain is a set of all pairs of layers and tracks. It is simple to describe the problem using a logic programming language like Prolog. However, as the problem is NP-hard, we cannot expect to solve it using simple backtracking algorithms if the problem is big. Simonis [13] has applied CHIP [4], a CLP language, to the two-layer and three-layer channel routing problems where there is only one vertical layer involved. Nevertheless, it is difficult to solve general multi-layer channel routing problems in CHIP-like languages because domains of variables in such languages are restricted to simpele sets of integers.

In this paper, we present a program for the problem in Beta-Prolog [18], an extended Prolog that supports state tables, i.e., relations in which each tuple is given a truth value *true* or *false*. The problem is treated as a CSP. The domains of variables are represented as a state table whose tuples are all the combinations of nets, layers and tracks. Initially, all the tuples are set to be true, meaning that each net is possible to be placed on any track in any layer. In this representation, to select a track T in a layer L for a net N, we just need to select a tuple (N, L, T) from the state table whose state is true, and to exclude a pair (L, T) from the domain of a net N, we just need to set the tuple (N, L, T) to be false. By using the operations over state tables, the manipulations required on the domains and constraints are described straightforwardly. In addition, the heuristics adopted to order variables are similar to those used for solving general

CSPs [8].

In order to find high quality solutions, we adopt a parallel execution model for executing the program on multi-sequential computers. The experimental results for the Deutsch's difficult problem, a well used benchmark problem, are very encouraging.

In Section 2, we define the channel routing problem in detail. In Section 3, we describe the algorithm used. In Section 4, we survey briefly the features of Beta-Prolog. In Section 5, we describe the program. In Section 6, we describe the parallel execution model. In Section 7, we give the experimental results. In section 8, we compare our approach with other approaches and discuss the directions for improving the program.

2 Channel Routing

A *channel* consists of two parallel horizontal rows with terminals on them. The terminals are numbered 1, 2, and so on from left to right. A *net* is a set of terminals that must be interconnected through a *routing path*. The *channel routing problem* is to find routing paths for a given set of nets in a given channel such that no segments overlap each other, and the routing area and the total length of routing paths are minimized.

There are a lot of different definitions of the problem that impose different restrictions on the channel and routing paths. We consider the *dogleg-free multi-layer* channel routing problem which impose the following three restrictions. Firstly, every routing path for every net consists of only one horizontal segment that is parallel to the two rows of the channel, and several vertical segments that are perpendicular to the two rows. This type of routing paths is said to be *dogleg-free*. It has been known that dogleg routing problems can be transformed into dogleg-free routing problems easily (see for example [6]). Secondly, the routing area in a channel is divided into several pairs of layers, one called *horizontal layer* and one called *vertical layer*. Horizontal segments are placed in only horizontal layers and vertical segments are placed in only vertical layers. The ends of segments in a routing path are connected through *via holes*. There are several tracks in each horizontal layer. Minimizing the routing area means minimizing the number of tracks. Thirdly, no routing path can stretch over more than one pair of layers. Thus, for each net, we only need to determine the horizontal layer and the track for the horizontal segment. The positions for the vertical segments are determined directly after the horizontal segment is fixed.

For example, Figure 1 depicts a set of nets. The terminals on the ith column of the top and bottom rows are denoted as $t(i)$ and $b(i)$ respectively. Figure 2 depicts a four-layer channel and the routing paths for the nets.

N_1 = {t(2),t(5)}
N_2 = {b(1),b(6)}
N_3 = {b(2),b(4)}
N_4 = {t(3),t(9)}
N_5 = {b(3),t(4),b(5)}
N_6 = {t(6),b(7)}
N_7 = {t(7),b(11)}
N_8 = {b(8),b(10)}
N_9 = {b(9),t(10),b(12)}
N_{10} = {t(11),t(12)}

Figure 1: The set of nets in the example problem.

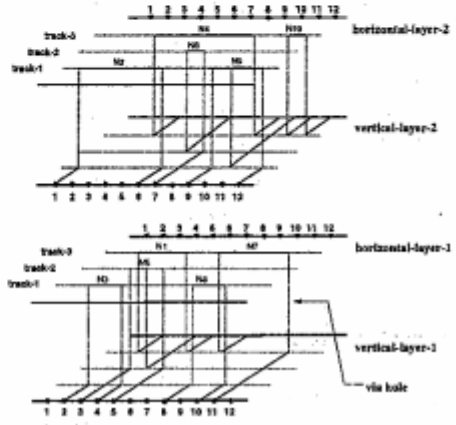

Figure 2: One solution for the example.

Two constraint graphs are created based on the given set of nets: one directed graph called a *vertical constraint graph* G_v and one indirected graph called a *horizontal constraint graph* G_h. In G_v, each vertex corresponds to a net and each arc from vertex u to vertex v means that net u must be placed *above* net v if they are placed in the same horizontal layer. The relation above does not necessarily reflect the physical configuration of tracks. The t_1th track in a layer is said to be above the t_2th track in the same layer if t_1 is greater than t_2. In G_h, each vertex corresponds to a net and there is an edge between two vertices u and v if net u and net v cannot be placed on the same track. Figure 3 depicts the constraint graphs for the set of nets shown in Figure 1. There is an arc from vertex 1 to vertex 5 in G_v because t(5) is included in N_1 and b(5) is included in N_5. If N_5 is placed above N_1 or on the same track as N_1 in the same horizontal layer, the vertical segments on the fifth column will overlap. There is an edge between vertex 1 and vertex 2 in G_h because the segment $(2,5)$ connecting the two farthest terminals in N_1 and the segment $(1,6)$ connecting the two farthest terminals in N_2 overlap each

other. Notice that the relation *above* is not transitive unless there is only one vertical layer in the given channel. For example, in a four-layer channel, N_4 can even be placed below N_8 in the same horizontal layer if N_9 is placed in a different horizontal layer.

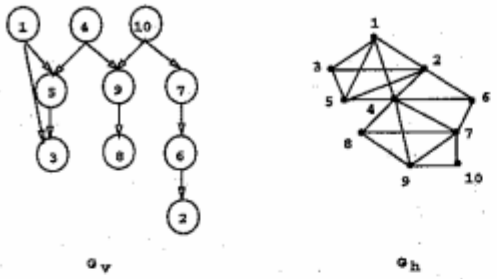

Figure 3: Constraint graphs.

The *depth* of a net u in G_v is computed as follows: If u lies at the top of G_v, then $u's$ depth is 0; otherwise, suppose u has n predecessors $v_1, v_2, \ldots, v_n$, then u's depth is $\max(\{d_{v1},\ldots,d_{vn}\})$ where d_v denotes the depth of v. There may exist cycles in G_v. In this case, all the vertices in a cycle have the same depth. The *length* of a routing path is the sum of the lengths of the horizontal and vertical segments in the path. For a horizontal segment whose left-most terminal number is l and whose right-most terminal number is r, the length of the segment is $r-l+1$. Let t be the number of tracks in each horizontal layer. The length of a vertical segment between the ith track and the top row is $t-i+1$ and the length of a vertical segment between the ith track and the bottom row is i.

3 Algorithm

We treat the channel routing problem as a CSP and use the forward checking algorithm [8, 15] to solve it. Each net is treated as a variable whose domain is the set of all pairs of layers and tracks. The constraints are represented by the two constraint graphs G_v and G_h.

The forward checking algorithm is a backtracking algorithm where constraints are used actively to reduce the search space. It solves a CSP by repeating the following steps until all the variables get values:

1. Choose a variable V that has not yet been assigned a value.
2. Select a value v for the variable V from its domain.
3. Exclude all those values from the domains of the remaining variables that cause inconsistency in the constraints after v is assigned to V.

When the domain of a variable becomes empty, the algorithm backtracks to the previous variable that has just been assigned a value. It undoes the assignment and assigns the next alternative value to the variable. If the next alternative value is not available, it continues to backtrack to the previous variable. If no variable exists to backtrack to, the algorithm terminates reporting a failure.

We use the following rules to choose a variable.

1. Choose first a variable whose corresponding net lies at the bottom in G_v.
2. Choose first a variable with the smallest domain.
3. Choose first a variable whose corresponding net has the greatest degree in G_v.
4. Choose first a variable whose corresponding net has the greatest degree in G_h.
5. Choose first a variable whose corresponding net lies at the deepest position in G_v.

The first rule ensures that the nets at the bottom of G_v are placed before those above them. All the other rules are consistent with the *first fail principle* [15]. Choosing first a variable that has the smallest domain and participates in the largest number of constraints can usually make a failure occur earlier.

4 Beta-Prolog

The novel feature of Beta-Prolog [18] is that it provides predicates for defining and manipulating state tables. In this section, we describe the motivations for introducing state tables into Prolog and the semantics of the predicates for state tables.

4.1 Motivations

Beta-Prolog is motivated by the following observations. Most combinatorial search problems can be formulated as state transition problems. Although Prolog is well used for search problems, it is unsatisfactory for solving many state transition problems due to its lack of appropriate data structures for representing states. An ideal data structure for representing states should meet the following three requirements: (1) it is fast to test conditions on states; (2) it is fast to update states; and (3) it is fast to backtrack to previous states.

In Prolog, a state can be represented as a list of facts that are true in the state. However, it is expensive to update or test the state. A state can also be represented as dynamic relations and updated using the predicates assert and retract. However, as updates performed by assert and retract are not undone upon

backtracking, we have to handle the restoration of the state in the programs explicitly. This will certainly make the programs obscure. In addition, accessing dynamic relations is not fast because the tuples in them are usually indexed only on the main functors of the first arguments. Another alternative is to represent a state as compound terms and use setarg, a predicate capable of updating a component destructively, to update the state. As the updates are undone upon backtracking, we need not handle backtracking explicitly in the programs. However, selecting a component from a compound term requires scanning the compound term.

4.2 State Tables

A state table is a relation in which each tuple is given a truth value *true* or *false*. It can be just considered as a relation of Prolog for which updates are backtrackable. When execution backtracks to a previous point, all updates on the state tables performed since that point are undone.

A state table or a part of a state table is declared by the following goal:

bt(p($X_1, X_2, \ldots, X_n$),S)

where p is an atom which denotes the name of the table, each $X_i (1 \leq i \leq n)$ is a *set expression*, and S is either *true* or *false* which is the truth value of the tuples. This declaration says that the Cartesian product $X_1 \times X_2 \times \ldots \times X_n$ is a part in the state table named p and all the tuples have the truth value S.

A set expression is a variable, an atomic term, a list of different atomic terms, a range $l..u$ or an interval (l,u). The range $l..u$ and the interval (l,u) both denote a set of consecutive integers l, $l+1$, ..., and u. Internally, the former is represented explicitly, while the latter is represented implicitly.

For example, the following call

bt(p([a,b],1..2),true)

specifies the state table shown in Figure 4.

A1	A2	State
a	1	true
a	2	true
b	1	true
b	2	true

Figure 4: Example state table.

4.3 Manipulations

The predicates on state tables take one or more *tuple patterns* and manipulate the tuples that match the patterns. A *tuple pattern* is a compound term in the form p(X_1,X_2,...,X_n) where p denotes the name of a state table, the first k (k≥0) arguments are atomic values or intervals and the remaining arguments are distinct variables.

The predicate **select(T)** selects a true tuple that matches the tuple pattern T nondeterminately. When the predicate **select(T)** is invoked, it selects the first true tuple matching T. When the execution backtracks to the predicate, it will select the next true tuple matching T automatically. When no true tuple exists, the predicate fails.

The predicates **first(T)** and **last(T)** get respectively the first and last true tuples that match the tuple pattern T. The predicates **prev**(T_1,T_2) and **next**(T_1,T_2) get respectively the previous and the next tuple T_2 of T_1. The predicates **true(T)** and **false(T)** test the truth value of the tuples matching the tuple pattern T. The predicates **set_true(T)** and **set_false(T)** update the truth value of the tuples matching the tuple pattern T. These updates are undone on backtracking. The predicate **count(T,N)** counts the number N of the true tuples that match the tuple pattern T.

For example, for the query

bt(p([a,b],1..2),true),select(p(b,X)).

X first gets 1. On backtracking, X gets 2.

5 Program

In this section, we present the program that specifies the forward checking algorithm for the channel routing problem.

The domains of variables are the sets of all pairs of layers and tracks. They are specified by a state table as follows:

bt(domain(1..N,1..L,(1,T)),true)

where N is the number of given nets, L the number of horizontal layers in the given channel, and T the number of tracks in each horizontal layer. Initially, all the tuples are set to be true, meaning that each net is possible to be placed on any track in any layer.

The variable for each net is represented as a compound term in the following form:

dvar(N,Layer,Track)

where N is the number of the net, *Layer* and *Track* are two variables that will hold respectively the layer and the track to be assigned to the net.

The constraint graphs are important both for choosing variables and excluding values from domains. We use two state tables called gv_above and gv_below to represent G_v. For every two nets u and v, if u is above v in G_v, then the tuples gv_above(u,v) and

```
route(N,L,T):-
      build_constraint_graphs(N),
      generate_vars(N,Vars),
      bt(domain(1..N,1..L,(1,T)),true),
      label(Vars),
      output(Vars).
label([]):-!.
label(Vars):-
      choose(Vars,dvar(N,Layer,Track),Rest),
      select(domain(N,Layer,Track)),
      update(N,Layer,Track),
      label(Rest).
```

Figure 5: Program for channel routing.

gv_below(v,u) are set to be true. G_h is represented by a state table called gh. For every two nets u and v, if there is an edge between u and v in G_h, then the tuples gh(u,v) and gh(v,u) are set to be true.

Figure 5 depicts the main part of the program. The first clause defines the predicate route(N,L,T) that routes a set of N nets on a channel with L horizontal layers each of which has T tracks. It builds constraint graphs, generates variables, builds the domains for the variables, calls label to assign values to the variables, and outputs the solution.

The predicate choose(Vars,Var,Rest) chooses a variable Var from $Vars$. All the information required for choosing variables is available from the four state tables: domain, gv_above, gv_below, and gh. For a net u, if first(gv_above(u,_)) fails, then u is at the bottom of G_v. The call count(gv_below(u,_),N) computes the degree of u in G_v, the call count(domain(u,_,_),N) computes the size of the domain of u, and the call count(gh(u,_),N) computes the degree of u in G_h.

The predicate update(N,Layer,Track), whose definition is shown in Figure 6, removes the net N from both G_v and G_h and excludes unsatisfactory values from the domains of the remaining variables. The call findall(A,select(gv_below(N,A)),As) finds all the nets As that are directly above N in G_v. The call update_v(N,As,Layer,Track) removes N from G_v by removing the arcs from every net in As to N, and ensures that no net in As can be placed below N in the same horizontal layer. The call findall(H,select(gh(N,H)),Hs) finds the neighbors Hs of N in G_h. The call update_h(N,Hs,Layer,Track) removes N from G_h by removing the edges between N and every net in Hs and ensures that no net in Hs can be placed on the same track as N.

```
update(N,Layer,Track):-
      findall(A,select(gv_below(N,A)),As),
      update_v(N,As,Layer,Track),
      findall(H,select(gh(N,H)),Hs),
      update_h(N,Hs,Layer,Track).
update_v(N,[],Layer,Track).
update_v(N,[A|As],Layer,Track):-
      set_false(gv_below(N,A)), % remove N from G_v
      set_false(gv_above(A,N)),
      set_false(domain(A,Layer,(1,Track))), % A is above N
      count(domain(A,_,_),Count),
      Count>0,
      update_v(N,As,Layer,Track).
update_h(N,[],Layer,Track).
update_h(N,[H|Hs],Layer,Track):-
      set_false(gh(N,H)), %remove N from G_h
      set_false(gh(H,N)),
      set_false(domain(H,Layer,Track)),
      count(domain(H,_,_),Count),
      Count>0,
      update_h(N,Hs,Layer,Track).
```

Figure 6: Update constraint graphs and domains.

6 Parallel Execution of the Program

In order to find good solutions, we adopt a parallel execution model similar to that proposed by Ali [1] and Lin [10] for runing the program on multi-sequential computers in parallel.

Before execution, each machine has a copy of the Prolog system and also the program to be executed. All the machines begin to execute the program from the same start point. When a machine reaches a goal select(T), it determines the part of the true tuples that will be treated as candidates for the tuple pattern T. All the tuples ignored will be explored by some other machines. This decision is based on a strategy known to all the machines.

Each machine keeps a list that contains the names of all the machines that are participating in the search of the same path. Initially, the list in every machine contains the names of all the machines, meaning that all the machines start from the same point. After reaching select(T), each machine deletes from the list the names of those machines that are assigned different tuples from that assigned to the machine.

When a machine reaches a goal select(T), it determines the tuples that will be treated as candidates for T. The scheduling strategy is based on the following three parameters: (1) the number of participating machines that will reach the same point; (2) the number of true tuples matching the tuple pattern T; and (3) the *parallel branching factor*, i.e., the maximum number of tuples that are explored in parallel at a

time.

A very simple scheduling strategy is adopted. Suppose the set of true tuples matching T is $\{t_0, t_1, \ldots, t_{n-1}\}$, the list of machines participating in the search of the same path is $m_0, m_1, \ldots, m_{l-1}$, and the parallel branching factor is b. Let k be the minimum of n and b. The machine $m_i (i = 0, \ldots, l-1)$ treats the following set of tuples as the candidates for T:

$$\{t_j \mid j \bmod l = i \bmod k \ (j = 0, \ldots, k-1)\}$$

If $k \geq l$, this strategy divides the k tuples evenly among the l machines; if $k < l$, this strategy divides l machines evenly among the k tuples. If $b < n$, then the tuples $\{t_b, \ldots, t_{n-1}\}$ will be rescheduled for the participating machines. For example, Figure 7 shows how a search tree is divided among four machines by the scheduling strategy. The parallel branching factor is assumed to be two.

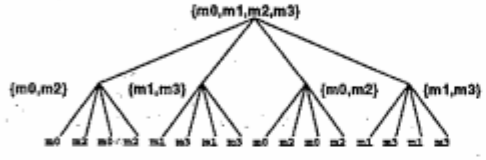

Figure 7: An example of parallel execution.

This execution model is very simple because neither communication nor copy is required. Unlike those models proposed by Ali [1] and Lin [10], this model place high priority on the left part of the search tree. For example, in Figure 7, the four machines will first concentrate on the left half part of the tree. If no solution is found, they move to the right part.

7 Experiments

In this section, we describe the experimental results for the Deutsch's difficult problem.

7.1 Environment and Benchmark

Beta-Prolog is an emulator-based Prolog system. The underlying abstract machine and the compiler are described in [17, 19]. In order to support the manipulations on state tables, we modified the trail stack such that when an update of a data cell needs be saved, both the address and the content of the cell are pushed onto the trail stack.

The program is of about 300 lines long excluding comments, blanks, the data for the nets, and the code for displaying a solution. It can minimize the number of tracks and the total length of routing paths by using branch & bound.

Order	HV(40)	2HV(20)	3HV(14)
1-2	31	14	9
1-3-4	34	13	9
1-5	29	13	8
1-2-3-4	30	13	8
1-5-3-4	28	13	8

Table 1: The best results found by sequential execution in 30 minutes.

The Deutsch's difficult problem is used as the benchmark problem. The benchmark suite described in [16] are well used in the VLSI design community, among which the Deutsch's difficult problem is a representative one. The problem is to route a set of 72 nets on a channel where there are 174 terminals on each row. There are 117 arcs in the constraint graph G_v and 846 edges in G_h.

7.2 Sequential Execution

We first run the program on a single SPARC-10 using different combinations of the heuristics listed in Section 3. Table 1 shows the best results found in 30 minutes (CPU time). The column **Order** denotes the combination of the heuristics used. The columns **HV**, **2HV** and **3HV** denotes the numbers of tracks in each horizontal layer for two, four and six layer channels respectively. The numbers in the parentheses denote the initial bound of the number of tacks.

The last combination 1-5-3-4 of heuristics shows the best performance. It found the best solution for HV that requires 28 tracks, which is known to be optimal, in only 19 seconds (see Figure 8). The CHIP program described in [13] found an optimal solution for the same problem in less than 30 seconds. The best result found for 2HV requires 13 tracks and that for 3HV requires 8 tracks in each horizontal layer. The optimal numbers of tracks for 2HV and 3HV are not yet known. The router described in [5] found in less than one second a solution for 2HV that requires 10 tracks, but it does not require segments in a routing path to be in only one pair of layers.

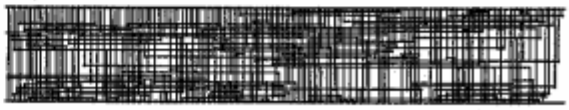

Figure 8: A solution found in 19 seconds.

Order	HV	2HV	3HV
1-2	30	13	8
1-3-4	33	12	8
1-5	29	12	8
1-2-3-4	30	12	8
1-5-3-4	28	12	8

Table 2: The best results found by parallel execution in 30 minutes.

7.3 Parallel Execution

We then run the program on 16 SPARCs using the same combinations of heuristics. Table 2 shows the best results found in 30 minutes. The best result found for 2HV requires 12 tracks, one less than that required by the solution found by sequential execution.

We finally run the program on 16 SPARCs to minimize the total length of routing paths. The heristics 1-5-3-4 is used and the time limit is set to be 30 minutes. Figure 9 depicts the best solution for 2HV.

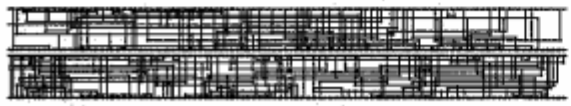

Figure 9: The best 2HV solution (length=4164)

8 Discussion

In this section, we compare our approach with other approaches and point out some directions for improving the approach.

8.1 Comparison with Traditional Algorithmic Approaches

There are a huge number of algorithms proposed to solve the channel routing problem. Recent algorithms tend to be very complicated and thus are very difficult to implement. Furthermore, when the restrictions on the channel or routing paths change, the algorithms must be redesigned. Compared with these traditional algorithmic approaches, the logic programming approach presented in this paper is very simple. It treats the channel routing problem as a CSP and solves it by using the forward checking algorithm. We can implement new heuristics for ordering variables by modifying the definition of the choose predicate and adapt the algorithm to other types of routing problems by modifying the definitions of domains and the update predicate. The state tables provided by Beta-Prolog makes it very easy to implement the algorithm.

8.2 Comparison with CLP Approaches

Because the channel routing problem can be treated as a finite-domain CSP, one would naturally think of using CLP languages like CHIP to solve the problem. The advantage of this approach is that the programmer does not need to program the procedure for updating domains explicitly. Simonis [13] has applied CHIP to the two-layer and three-layer channel routing problems where there is only one vertical layer involved. The relation *above* is represented as disequalities. However, as we have described in Section 2, the relation *above* is not transitive for general multi-layer channel routing problems and thus cannot be represented as disequalities.

The formulation described in this paper does not suit CHIP because the domains in CHIP are restricted to sets of integers. To make the formulation suitable to CHIP, we can use $l+1$ variables L_u, T_{u1}, T_{u2}, ..., and T_{ul} for each net u where l is the number of horizontal layers, L_u holds the layer, and each T_{ui} $(i = 1, 2, \ldots, l)$ holds the track in the ith layer. Because each net is placed in only one horizontal layer, only one variable in T_{u1}, T_{u2}, ..., and T_{ul} will be instantiated. For two nets u and v, if u is above v in G_v, the following implication constraints must be satisfied:

$$L_u = L_v = 1 \rightarrow T_{u1} > T_{v1}.$$
$$\ldots$$
$$L_u = L_v = l \rightarrow T_{ul} > T_{vl}.$$

It is possible for the constraint solver to reduce the domains of the variables on the right-hand sides if the left-hand sides of the implication constraints become true. However, checking constraints is expensive because a constraint need be checked every time when a participating variable in it is instantiated.

8.3 Improvements

The program can be improved in several directions. Firstly, the current program only uses general heuristics for ordering variables. It would be more efficient to use some problem specific heuristics used in traditional algorithms. For example, such information about nets concerning the lengths of nets, types of nets (two-terminal nets, multi-terminal nets, nets connecting only terminals at the top, nets connecting only terminals at the bottom, etc.) can be used to choose variables. Secondly, the program can be improved by introducing heuristics for choosing appropriate values for selected variables. These two improvements should be justified by experiments. For this purpose, a large number of benchmark problems must be tested.

Thirdly, on a network of computers, much better solutions can be obtained if computers are able to communicate each other. With communication, a global bound can be established for branch & bound search which is used in the program to minimize the total length of routing paths.

9 Conclusion

In this paper, we showed how to solve the multilayer channel routing problem using Beta-Prolog, an extended Prolog with state tables. The problem is treated as a CSP, and the domains and constraints are represented as state tables. We also showed how to execute the program in parallel on multi-sequential computers. The results obtained for the Deutsch's difficult problem are very encouraging.

Acknowledgement

The author is grateful for Dr. Moon R. Jung for his valuable comments on the presentation. This research was partially supported by a research fund (No.06750395) of the Ministry for Education, Science, and Culture of Japan.

References

[1] Ali, K.A.M.: OR-Parallel Execution of Prolog on a Multi-Sequential Machine, Int. J. Parallel Program., Vol.15, No.3, 189-214, 1986.

[2] Burstein, M: Channel Routing, Layout Design and Verification, North-Holland, pp.133-167, 1986.

[3] Deutsch, D.N.: A Dogleg Channel Router, Proc. 13th Design Automation Conference, pp.425-433, 1976.

[4] Dincbas, M., Van Hentenryck, H., Simonis, H., Aggoun, A., Graf, T., and Berthier, F.: The Constraint Logic Programming Language CHIP, Proc. FGCS'88, 693-702, 1988.

[5] Fang, S.C., Feng, W.S., and Lee, S.L.: A New Efficient Approach to Multilayer Channel Routing Problem, Proc. of the 29th ACM/IEEE Design Automation Conference, 579-584, 1992.

[6] Gao, S. and Kaufmann, M.: Channel Routing of Multiterminal Nets, J. ACM, Vol.41, No.4, pp.791-818, 1994.

[7] Hashimoto, A and Stevens, S.: Wire Routing by Optimizing Channel Assignment within Large Apertures, Proc. 8th Design Automation Workshop, pp.155-169, 1971.

[8] Kumar, V.: Algorithms for Constraint Satisfaction Problems: A Survey, AI Magazine, Spring, 1992.

[9] LaPaugh, A.S.: Algorithms for Integrated Circuits Layout: An Analytic Approach, PhD Dissertation, MIT Lab. of Computer Science, 1980.

[10] Lin, Z.: Self-Organizing Task Scheduling for Parallel Execution of Logic Programs, Proc. FGCS'92, 859-868, ICOT, 1992.

[11] Liu, X., Sakamoto, A. and Shimamoto, T.: Genetic Channel Router, IEICE Trans. Fundamentals, E77-A, pp492-501, 1994.

[12] Mackworth, A.: Constraint Satisfaction, Encyclopedia of Artificial Intelligence, John Wiley & Sons, pp.205-211, 1986.

[13] Simonis, H.: Channel Routing Seen as a Constraint Problem, Tech. Rep., TR-LP-51, ECRC, Munich, July, 1990.

[14] Takefuji, : Neural Network Parallel Computing, Kluwer Academic Publishers, 1992.

[15] Van Hentenryck, P.: Constraint Satisfaction in Logic Programming, MIT Press, 1989.

[16] Yoshimura, T. and Kuh, E.S.: Efficient Algorithms for Channel Routing, IEEE Trans. CAD, Vol.1, pp.25-35, 1982.

[17] Zhou, N.F.: Global Optimizations in a Prolog Compiler for the TOAM, J. Logic Programming, Vol.15, pp.265-294, 1993.

[18] Zhou, N.F.: Beta-Prolog: An Extended Prolog with Boolean Tables for Combinatorial Search, Proc. 5th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society, pp.312-319, 1993.

[19] Zhou, N.F.: On the Scheme of Passing Arguments in Stack Frames for Prolog, Proc. Eleventh International Conference on Logic Programming, MIT Press, pp.159-165, 1994.